Reusing and Combining UI, Task and Software Component Models to Compose New Applications

Christian Brel
CNRS, Laboratoire I3S - UMR 7271 - UNS CNRS
FR-06900 Sophia Antipolis Cedex
Christian.Brel@unice.fr

Philippe Renevier Gonin
Universit Nice Sophia Antipolis,
Laboratoire I3S - UMR 7271 - UNS CNRS
FR-06900 Sophia Antipolis Cedex
Philippe.Renevier@unice.fr

Alain Giboin
INRIA, Laboratoire I3S - UMR 7271 - UNS CNRS INRIA
FR-06900 Sophia Antipolis Cedex
Alain.Giboin@inria.fr

Michel Riveill
Universit Nice Sophia Antipolis,
Laboratoire I3S - UMR 7271 - UNS CNRS
FR-06900 Sophia Antipolis Cedex
Michel.Riveill@unice.fr

Anne-Marie Dery
Universit Nice Sophia Antipolis,
Laboratoire I3S - UMR 7271 - UNS CNRS
FR-06900 Sophia Antipolis Cedex
Anne-Marie.Pinna@unice.fr

Composing applications by considering in parallel both software components and UI elements is a complex process not yet very well supported by any current composition process model or composition environment. To contribute to better support the composition process, we propose a new composition model and a prototype of a component assembler, the so-called OntoCompo, which implements the model. The model describes applications in terms of Task, UI and software components. The prototype allows a composition mainly driven by the direct manipulation of UI elements, the other components being hidden, but still being linked to the UI elements. We performed a user testing with actual developers to evaluate if the composition process was actually facilitated by our modeling approach and the prototype implementing it.

UI Composition ; Model-Based Development ; User Testing

1. INTRODUCTION

Facilitating the development work of software developers was the motivation of the OntoCompo approach reported in this paper. We could call this approach the *Developer-Friendly Development* (DFD) approach, by comparison to the so-called *End-User Development* (EUD) (Lieberman et al. (2006)). EUD has been defined ”as a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artefact”. In 2006, (Lieberman et al. (2006)) foresaw that ”the goal of human-computer interaction will evolve from just making systems easy to use (...) to making systems that are easy to develop”, implied end-users to develop. Our DFD approach aims at making application composition easier for professional developers themselves.

Composing applications, by considering in parallel both software component assembly and User Interfaces (UI), is a complex process not yet very well supported by any current modeling approach or composition environment. The need to combine applications may grow with the increase of specialized applications available on application stores. For example, Google Maps is often integrated for geo-localization. In an idealistic way, developers must be able to reuse existing functionalities with minor developments. There is a need in supporting developers in their task of combining elements of existing applications to create a new application.

Rather than having to learn API of applications such as Google Maps and to code from scratch, or rather than having to abstract existing applications (in terms of tasks, services or UI) and to transform such abstractions in some new application, we propose a new composition model and a prototype, the so-called OntoCompo, to simplify the composition work. The model describes applications in terms of *links* between *Task*, *UI* (i.e., graphical elements) and *Software Components*. By preserving and by reasoning on these links, during developers' direct manipulations on the UI, we intend to enhance the consistency of the composition. In our case study, we consider the UI elements as an entry point. Developers can indicate directly on the UI which visible parts of the application are required for the composition. By considering at the same time UI, task and software models, i.e., by following the linked UI, Task and Software descriptions, we could transfer such UI direct manipulations to the whole application description.

We assumed that hiding the three models would hide a part of the complexity of the process, and consequently facilitate it. To evaluate if the composition process was actually facilitated by our approach through the prototype implementing it, we performed a user testing with actual developers.

This article is structured as follows. After having presented related work, we describe our application model for composition. Next we present the OntoCompo prototype. Then we report the user testing (method, results and discussion). Finally we conclude with future works.

2. RELATED WORK

2.1. Software Composition

For software composition, "Composition can be defined as any possible and meaningful interaction between the software constructs involved" according to (Lau and Rana (2010)) where a taxonomy of composition mechanisms (e.g., orchestration, aspect oriented programming, etc.) is defined. When the application code is not available, we can only access to published interfaces and we have to use connectors (Mehta et al. (2000)).

2.2. UI Composition

2.2.1. UI composition approaches

We distinguish two different UI composition approaches. *(1)* The *first approach* bases the UI composition on abstract description, like in UsiXML (Lepreux et al. (2010)), the ServFace project (Paternò et al. (2011)), Alias (Joffroy et al. (2011)) and Transparent Interface (Ginzburg et al. (2007)). Those models are defined by XML languages. Final UIs are obtained thanks to model transformations. *(2)* The *second composition approach* is based on "UI Components", which are reusable high-level widgets, available in repositories. Compose (Gabillon et al. (2011)), COTS-UI (Criado et al. (2010)), CRUISe (Pietschmann et al. (2009)), WinCuts (Tan et al. (2004)), UI façades (Stuerzlinger et al. (2006)) and on-the-fly mashup composition (Zhao et al. (2008)) illustrate such composition. Several of these works, e.g., (Nestler et al. (2009); Gabillon et al. (2011)), express and manipulate requirements with tasks.

2.2.2. Models used in UI composition

Three kinds of models are used in the two approaches: *Task models* (e.g., trees of users' tasks), *UI models* (e.g., hierarchies of graphical elements), and *Software models* (e.g., assemblies of components). Table 1 reports the kinds of models used as entry point (i.e., the models manipulated in order to drive the composition) in the related work we analyzed. The last row of the table represents the characteristics we wanted to include in our approach: we wanted *(1)* to use the three kinds of models in order to make the composition easier, and *(2)* to reuse existing applications in order to avoid re-designing what has been already designed. Requirement (1) led us to rely on the existing approaches and/or systems considering several kinds of models instead of only one model.

ServFace (Paternò et al. (2011); Nestler et al. (2009); Paternò et al. (2009)) and Service-Interaction Descriptions (Vermeulen et al. (2007)) start with building a new Task Tree, associate Service to tasks, then produce a new UI by assembling abstract UI fragment associate to services and finally complete the new UI. In those works, the composition produce a new UI and is based on an abstraction of the wished composition.

On-the-fly service composition (Zhao et al. (2008)), COTS-UI (Criado et al. (2010)) and CRUISe (Pietschmann et al. (2009)) compose web services or software components and their UI, but without considering tasks.

Compose (Gabillon et al. (2011)) starts with the translation in term of tasks of a requirement express in natural language and then a new UI is computed. Compose is designed for end users in a context of UI adaptation while our context is application designs made by developers.

2.3. Implications for Our Approach

From the analysis of these works, we noted that we can compose the UI (respectively, the functional parts) of former applications, but that we must build

Related Work	Entry Point			Used Models			Results	
	S[1]	Task	UI	S[1]	Task	UI	Reuse	G[2]
Service Approach: BPEL4WS (Khalaf et al. (2003)) - BPEL (Alves et al. (2007)) - Web Service Composition OWL-S (Sohrabi and McIlraith (2010))	x			x			x	
Component Approaches : Fractal (Bruneton et al. (2006)), SCA (et al. (2005); Ope (2007)) and SLCA (Hourdin et al. (2008))	x			x			x	
ALIAS (Joffroy et al. (2011)) ; Transparent interface composition (Ginzburg et al. (2007))	x			x		x	serv[4]	UI
ServFace (Paternò et al. (2011); Nestler et al. (2009); Paternò et al. (2009)) ; Service-Interaction Descriptions (Vermeulen et al. (2007))	x	x		x	x	x	serv[4]	UI
on-the-fly service composition (Zhao et al. (2008))	x		x	x		x	x	
Task Composition (Bourguin et al. (2007))		x		x	x			code[3]
Task Tree Merge (Lewandowski et al. (2007))		x		x	x		x	
Compose (Gabillon et al. (2011))		x		x	x	x	x	
ComposiXML (Lepreux et al. (2010))			x			x	UI	
Migratable UI (Luyten et al. (2002))			x		x	x	CUI	
WinCuts (Tan et al. (2004)) ; UI façades (Stuerzlinger et al. (2006))			x			x	x	
COTS-UI (Criado et al. (2010))			x	x		x		x
CRUISe (Pietschmann et al. (2009))	x		x	x		x	x	
OntoCompo Expected Features	***x***	***x***	***x***	***x***	***x***	***x***	***x***	

[1]Software Component [2]Generation [3]code to be completed [4]services

Table 1: *Classification of Composition Approaches*

again the functional parts (respectively, the UI). Moreover, none of these works allows reusing former applications and supporting replacement of UI parts. Because of our two aims, easing the composition and reusing former part of applications, we opted for a component-based modeling linked with a UI model and a Task model. The composition will be performed by transforming the manipulation on models to manipulation on components.

3. APPLICATION MODEL FOR COMPOSITION

OntoCompo exploits the three points of view for application composition identified in the state of the art, i.e., the three descriptions: UI, Tasks and Software Component (Assembly). The originality of this approach is to connect each of those models with the two others (Brel et al. (2011)) by linking corresponding entities: *(1)* each graphical element is linked to tasks it supports; *(2)* each task is linked to graphical elements used to perform it; *(3)* each graphical element is linked to the software component surrounding it; *(4)* each software component is linked to graphical element surrounded by it ; *(5)* each task is linked to software component used to performed it; *(6)* each software component is linked to tasks it supports. Those links are expressed by annotations on the three models.

We model UI with a classical hierarchical description of graphical elements. That description is annoted with layout links such as *on the left*, *below*, etc. We model Tasks as ConcurTaskTrees (CTT) (Paternò et al. (1997)). We model Software Assembly as Component Assembly. Components are connected through their ports. A connection between two components is between a requiring port and a providing port. So ports are characterized by their *nature* (”**required**” or ”**provided**”), their *type* (”**trigger**” for activating actions, ”**input**” for entering or providing values or ”**output**” for displaying or storing data), and their *role* (”**UI**” when concerning the graphical interface, ”**UI component**” when concerning the manipulation of graphical element, etc.). The links with software components and tasks or graphical elements are done on their ports. Figure 1 illustrates the application model for composition. For example, the text entry ”AddressAInput” is connected with the task ”Fill Position A” and with the two ports of the software component ”AddressAInput”: one, with the port tagged ”UI”, for getting the typed value and the other, with the port tagged ”UI Component”, for graphical element exchange between software components in order to build the UI.

The approach is applied to the application composition driven by UI manipulation. Thus, starting from a selected part of the UI, corresponding software components are identified. The connections between models are exploited in a process of selection, composition by substitution and layout reorganization.

1. Selection consists in selecting the parts of UI required for the composition. Thanks to the UI model, the selection is completed in order to obtain an operational and usable selection. Moreover, thanks to extensions (query based on the models), the selection is eased by following the connection.
For example, all graphical elements required for achieving a parent task can be retrieved. Let consider the selection of the graphical element "AddressAInput". The latter is connected with the task "Fill Position A" which parent task is "Fill begin and arrival position" (FBaAP). We extend the selection of graphical elements to all graphical elements connected to (at least) the task "FBaAP" or one of its subtasks. We obtain a set of graphical elements composed of "AddressAInput" and "AddressBInput". Then, we consider all software components connected to (at least) one of these graphical elements or to the task "FBaAP" (or to its substasks). If there are unsatisfied required ports in the selection, corresponding missing software components and graphical elements connected with such missing components are added to the selection, until there is no unsatisfied required port left. The final selection is $\{$ {AddressAInput, AddressBInput}$_{UI}$; {"Fill begin and arrival position}$_{task}$; {AddressAInput, AddressBInput}$_{SC}$ $\}$.

2. Composition by substitution allows to replace a selected component by another "equivalent" component. The substitution is based on the connections between software components. Thanks to the characterization (nature, type, role) of component ports involved in connection, the substitution is realized by the addition of adapters between already connected ports or between ports to connect. So the links between the initial applications are set up to create the new application, by accordingly modifying the software component assembly (Brel et al. (2012)).
In the context of application composition driven by UI manipulation, to perform such substitution, actions on graphical elements are propagated to corresponding software components. To choose how to make a substitution, i.e., which port must replace which port, the approach allows to apply different strategies: asking for the user, applying an algorithm, etc. In our approach, we generate skeleton of adapters whose code needs to be completed.

3. Layout reorganization is a simple step where remaining selected UI elements are placed in the windows. Once the placement done, the final application is generated: a new component assembly is produced and also the skeleton of adapters. A developer has to fill the content of those adapters in order to finalize the composition.

The model for application composition enables several possible interactions. Wanting to determine the better way to apply the approach, we developed OntoCompo in order to experiment a simple use of the model in a composition process. OntoCompo sequentially implements the "selection-substitution-layout reorganization" process. OntoCompo also hides underlying models to the developer performing a composition. The developer only manipulate graphical elements. In the next section, we present an implementation of OntoCompo.

4. IMPLEMENTATION OF ONTOCOMPO

Our application models are developed thanks to ontologies, allowing us to quickly perform the necessary requests for composing. OntoCompo (Brel et al. (2011)) manipulates applications with Fractal[1] components (Bruneton et al. (2006)), which must be semantically described. To implement our functions and algorithms, we make SPARQL requests, processed by the semantic engine CORESE / KGRAM (Corby et al. (2012)).

The initial applications are developed according to component architecture, defined by the Julia implementation (in Java) of the Fractal model. The whole application, whether its features or its graphical interface, is implemented by components. Some components encapsulate graphical elements (from the SWING library of java) and are recognizable by their particular ports with the "UI Component" role.

The architecture of OntoCompo consists of three interrelated parts (see Figure 2): (1) The *Application Loader*, for loading software components and models (semantic descriptions); (2) the *OntoCompo GUI*, implementing the three application composition steps; and (3) the *OntoCompo API*. This API is the main part of the architecture; it handles all the manipulations to be made by our algorithms on fractal components or semantic descriptions. Through a collection (a map) allowing to associate a Swing component with the encapsulating fractal component, the API can retrieve and manipulate fractal components from selected graphical elements given by the OntoCompo GUI.

A video illustrating OntoCompo and the scenario used in the experimentation is available at http://goo.gl/QEqf4g .

5. USER TESTING OF ONTOCOMPO: METHOD

To validate our approach of the application composition driven by the manipulation of UI

[1]This software component model was named "Fractal" because its components could be made of several components.

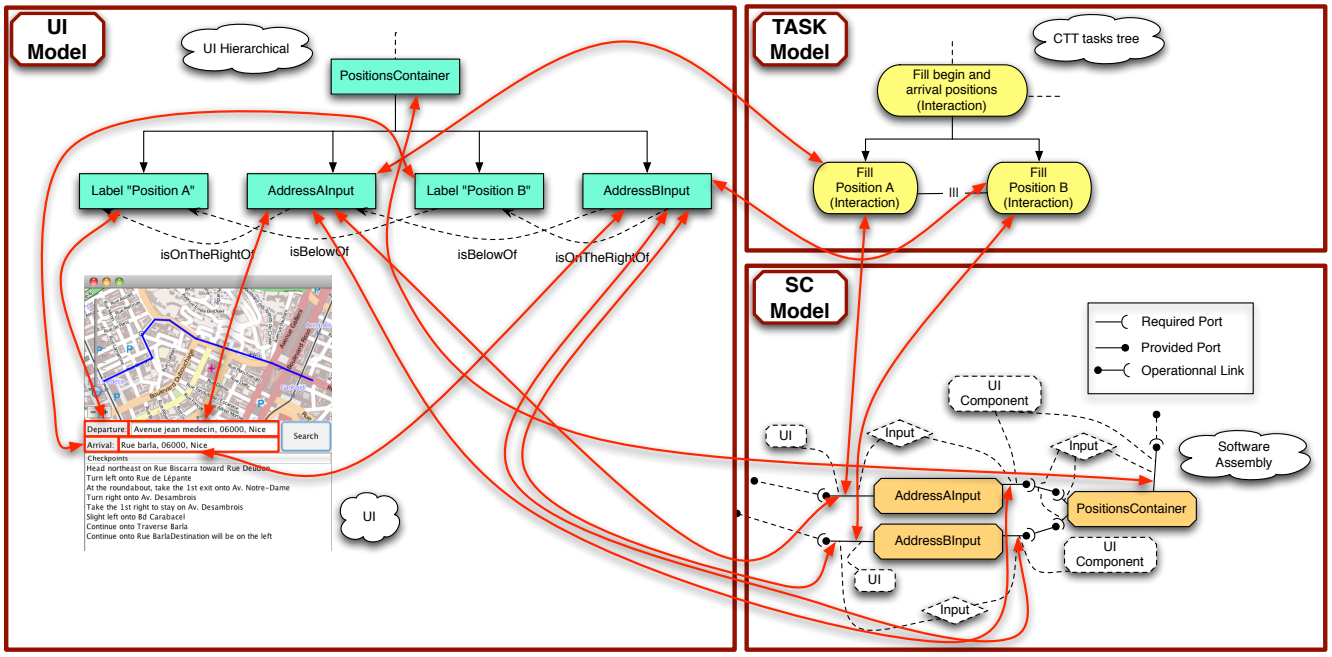

Figure 1: *Excerpt of the application model linking UI, Tasks and Software Components (SC).*

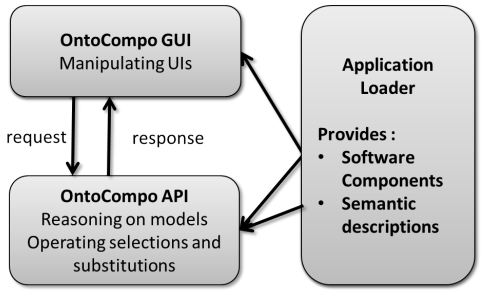

Figure 2: *Architecture of the OntoCompo prototype.*

elements, we performed the following user testing with actual developers.

5.1. Evaluation Method Type

Quantitative methods are classically preferred over qualitative methods to evaluate systems. For example: (a) evaluating their UsiXML-based composition tool GrafiXML, (Lepreux and Vanderdonckt (2007)) used the GOMS method to establish the time gain during the use of GrafiXML; (b) user testing their platform for migrating interface components to a target device, (Paternò et al. (2011)) submitted a quantitative questionnaire (with Likert scales) to the users of the platform. In our case, using strictly quantitative methods to evaluate our approach and its supporting tool was considered premature. We had to *qualify* developers' actual practices of composition before to *quantify* them. Hence we needed a method that was both qualitative and quantitative. We used a variant of the "cooperative evaluation" method of (Monk (1993)).

5.2. Goal and Hypotheses

Our goal was to evaluate the understanding and acceptance of the OntoCompo approach as performed through the OntoCompo tool. We envisioned two working hypotheses:

- *Strong hypothesis:* Developers can perform their composition task by manipulating graphical elements only; any of the three models (UI model, Task model and Software-Components model) is necessary. Manipulating the code of the resulting application is not necessary either. This hypothesis reflects the ideal we sought to facilitate the composition work of the developer.
- *Weak hypothesis:* To perform their composition task, in addition to graphical elements, developers need to manipulate the three kinds of models, but to different degrees. Manipulating the code of the resulting application remains not necessary. This hypothesis means that development work facilitation is variable.

5.3. Participants

Two kinds of developers participated to the user testing: four developers who never handled an application composition tool; five developers who already used some composition tool (not necessarily a tool for composing applications). Since no

differences in their behavior were noticed, we decided to consider them as a single group.

5.4. Material

The material used during the experimentation was: (a) the OntoCompo Prototype; (b) the composition task instructions; (c) the "additional information" to be provided on demand to developers or during the debriefing phase; this information consisted of printed documents representing: the Task Model, the Software-Component Model, the UI Model, and the generated code; excerpts of the models are given in Figure 1; and (d) the composition task scenario, including the UIs of the two source applications and the UI of the resulting application (described in detail in the next subsections).

5.4.1. Composition task scenario and related UIs

Composition task scenario: "A developer has at her disposal two applications: one, called "Movie Theaters", for displaying movies played in cinema near a specified location, and another, called "Maps", for searching directions on a map. She wants to produce a new application to search movie theaters closed to a specified address. Once a movie theater is selected, then the directions from that address to the selected theater are displayed."

The "Movie Theaters" UI includes at the top of the window a text field to enter the address that will be the geographical center of the research. The application uses a Web Service form the Web Site http://www.allocine.com, a French web site about movies and theaters. The Web Service enables queries to find theaters from a location, to list the movies played in a theater, etc. The user calls that service by clicking the "search" button. The table on the left of the window is fullfilled with the received answers. By clicking on a line of that table, another query is made to the Web Service to get the list of played movies with their showtimes. The returned list is displayed in the table on the right of the window.

The "Maps" UI is vertically organized. At the top, there is the panel for displaying the map. At the middle, there is the form to fill the start and the arrival. At the bottom, there are the main intersections of the found route.

The resulting UI: an example of the UI resulting from the composition scenario is given in Figure 3. Only one text field is left, to enter the start of the route for the "Maps" application and to enter the geographical center of the search for the "Movie Theaters" application. From the latter, only the list of the closed theaters is displayed. It is only after selecting one theater that the route is displayed on the maps, coming from the "Maps" application.

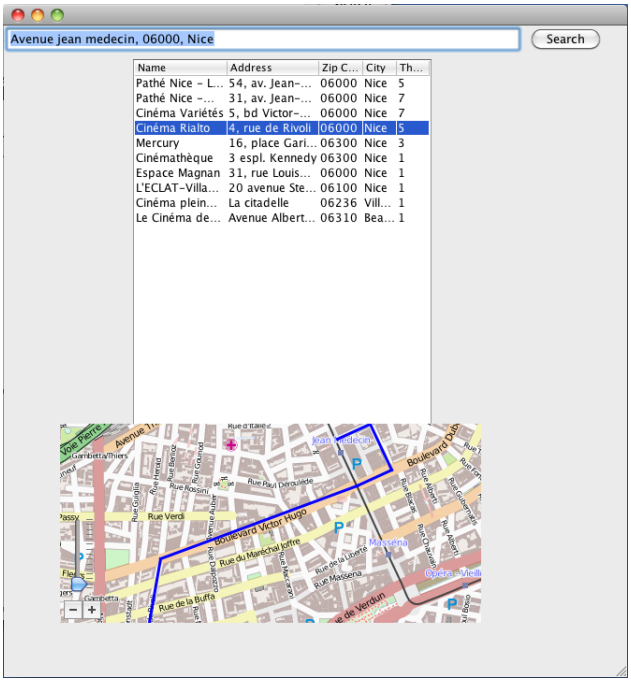

Figure 3: *UI of the application resulting from the composition scenario.*

5.4.2. Extensions of selection

During the selection step, the developers can use consistent extensions of selection thanks to queries in all models, as described in Section 3.

5.4.3. Scenario difficulties

To make the scenario realistic, we included three main difficulties for the substitutions in the application composition task of the developer.

Difficulty 1: A misleading similarity of two graphic elements. Two buttons present in both applications had the same shape and the same title "Search". This similarity can mislead the developer who may be tempted to merge them, a merging that is not in the composition scenario. One button (from "Movie Theaters") must be kept, the second one (from "Maps") must be substituted by a trigger associated with the selection of one theater in the list.

Difficulty 2: A substitution of two heterogeneous graphic elements. The first element, to be kept, was a list of the closest cinemas in the application "Movie Theaters". The second element, to be replaced, was an entry text (allowing to inform the address of arrival necessary for the calculation of the route in the application "Maps"). The list is obviously an "Output", and the entry text an "Input". The substitution was possible because the software element associated with the list supplies a port of type "Input" to obtain the selected cinema.

Difficulty 3: A generated adapter source code (to be completed) including two methods with the same signature. This difficulty comes from the code presented to the developer after a substitution. It indicates the same method signature twice. The adapter generated by the substitution has several ports. Each of them corresponds to an implementation of a software interface. In the adapter source code, since two different software interfaces include the same method signature, there is twice the same signature of method $public\ String\ getInput()$. Even if this difficulty comes from the language Java, which does not allow to make the difference between both methods of the same signature resulting from two different software interfaces, we expected that the developer would know how to react. Indeed in this example, both methods $getInput()$ have to produce the same result: merging the two methods is here possible.

5.5. Procedure

Each developer was placed in front of the OntoCompo prototype, next to the experimenter leading the developer's testing session. As the developer went along, additional information was given to her on demand. Explanations on the use of the prototype were also given by the experimenter when requested by the developer. Each session consisted of three phases: **(1) a familiarization phase** where the developer freely manipulated the prototype interface to become familiar with it; **(2) a task-performance phase** where the developer performed the substitution task proposed by the experimenter; at the end of this phase, the developer was presented with the code generated on the outcome of the composition. The developer was expected to understand and explain that this code corresponded to an adapter generated during substitutions; **(3) a debriefing phase** where the developer provided further feedback.

5.6. Data Collection and Analysis

5.6.1. Data collection

The manipulation of OntoCompo was video-recorded. Oral exchanges between the developer and the experimenter were also recorded. An observer was sitting back the developer to take notes on the developer's behavior. The experimenter also took notes when possible for him. Data collected consequently were: experimenter's and observer's written notes; videos; developers' verbalizations.

5.6.2. Data analysis

The analysis consisted in determining if the developers achieved each composition task (or stage of the process: selection, substitution, layout reorganization) using graphical elements only (*Strong Hypothesis*), or if they needed to rely on the UI, Task and Software Components models (*Weak Hypothesis*), i.e., if they asked for the additional documentation representing these models.

Extension Type	Use	Asked Information
UI	44% (4/9)	No
Task	67% (6/9)	Yes (for 5 developers) No (for 1 developer)
Software	11% (1/9)	Yes

Table 2: *Extension uses during user tests.*

Performing at least 1 section with n extensions			
n=0	**n=1**	**n=2**	**n=3**
33% (3/9)	78% (7/9)	22% (2/9)	0%

Table 3: *Proportion of extension uses.*

6. USER TESTING OF ONTOCOMPO: RESULTS AND DISCUSSION

On one hand, the nine developers well understood the composition process, and succeeded in manipulating the tool and in performing what they planned; however, only 55% (5/9) succeeded in making the composition without error. On the other hand, it emerges that additional information helped most developers (67%; 6/9) to achieve the composition.

The results are essentially qualitative. In order to present the experimentation result, we summarized developers' comments and feelings. For example, 67% of developers needed additional information means that by analyzing the 9 experimentation sessions, we found that 6 of them required more information, either thanks to their comments or thanks to their used of printed additional information.

6.1. Developers' General Performance

6.1.1. How extensions were used

Table 2 summarizes the uses of extensions. The test containing several selections, the developers varied in the use of the extensions. They made sometimes at least a selection without extension, sometimes with two extensions, and often with one extension (including combination of several extensions applied at the same time). They had several opportunities to use one or several extensions, so the percentage accumulation is upper to 100% in Table 3.

We notice that 44% (4/9) of the developers used a combination of two extensions, in particular task with software extensions, e.g., extending the selection by the software component links while preserving only elements involved in the same task.

6.1.2. How scenario difficulties were addressed

Difficulty 1 (close resemblance of two "Search" buttons) confused several developers. A difference exists clearly between what we had considered as manipulations before performing the tests and the manipulations which the developers effectively made. This difficulty, for 44% (4/9) of the developers, led them to merge both buttons directly even though this substitution was identified as not necessary.

Difficulty 2 (substitution of two heterogeneous elements, a list and an entry text) led most developers to substitute not the list with the entry text but rather with an element of the list (among others, the address of the selected cinema). Yet, this action was not allowed by the approach because each modeled graphical element was considered as indivisible. The expected substitution was always achieved, but 44% (4/9) of the developers strongly hesitated and achieved the right substitution after having eliminated the other possibilities.

Difficulty 3 (adapter source code including two methods with the same signature) was circumvented by developers by suppressing the redundant method, but without fully understanding why. This highlights a lack of information.

6.2. Developers' Need for Models

A general analysis underlined that additional information would ease the use of OntoCompo. The developers' preferences during the debriefing are summarized in Table 4. A majority of them (78% - 7/9) asked for an interactive representation of the task model during the phase of selection. This integration would allow to match the selection of the tasks with the associated graphical elements and vice versa. The task tree was indicated as the most intuitive model to analyze the behavior of the application especially if it allows to identify the correspondences between graphical elements and tasks. We also noted that 67% (6/9) of developers (cf. Table 2) having used the Task extension made an "abstract" deviation towards the software component model, deducting links between components exclusively from the information on the tasks. Moreover, the expressed preferences showed that for the substitution step, 67% of the developers wished an access to the software component models. On the contrary we noticed that no additional information was necessary for the use of UI Extensions. However 44% (4/9) of the developers (not necessarily the same that those who used this extension) expressed the fact that the representation of UI model seems necessary with more complex graphical interfaces. According to the developers, this model would highlight the information on the interweaving of UI elements.

6.3. Discussion

The results of this user testing are encouraging. Participant developers welcomed well our model of application and the composition process. They generally succeeded in realizing the expected application. Difficulties met by the developers are the most often related to the lack of information about the underlying models. The identified needs for additional information show that the strong hypothesis we formulated rarely correspond to the developers' practices. In other words, we have to say that the composition task can not be only driven by the manipulation of UI graphical elements. The weak hypothesis seems to be the most realistic: to perform their composition task, developers need to manipulate the three kinds of models together, but to different degrees. In other words, we have to say that the composition task must be driven by at least two of the three models (UI model, Task model, and Software Components model), depending on the process step. Results of the user testing revealed the most interesting models in the different steps, i.e., the additional information to be provided at these steps: **(1)** the **UI model and the Task model** are the most adapted **for selecting** the relevant part of applications. This can be explained because the two models are used to describe the interactions in user-centered design. **(2)** The Software **Components model**, and to a lesser extent the Task model, is the most adapted **for the substitution** step, because of the underlying impact on the software components.

Such additional information will help developers to predict and to explain both the result of a selection extension, and the possible substitutions. Such information must be be kept until the final application is generated in order to provide explanations to the developer when needed. Providing the developers with the underlying models would not only guide and reassure them, but also limit their cognitive load.

7. CONCLUSION AND FURTHER WORK

We have presented our approach of application composition based on three application descriptions or models, namely the UI, Task and Software Components descriptions or models. We described OntoCompo, the prototype applying our approach to an application composition driven by manipulation of UI graphical elements only. The user testing we performed with OntoCompo highlighted that this restricted manipulation was not enough to achieve an appropriate composition. Results led us to conclude that the most realistic of our working hypotheses was the weak hypothesis, namely: to perform their composition task, developers need to manipulate the three kinds of models together, but to

	Selection	Substitution	UI Reorganization
UI	Information **needed** by 44% (4/9) of the developers (when the applications would be more complex)	Information **not needed**	Information **not needed** (needs in keeping information used during selection and substitution)
Task	Information **needed** by 78% (7/9) of the developers	Information **needed** by 22% (2/9) of the developers	Information **not needed**
Software	Information **needed** by 44% (4/9) of the developers	Information **needed** by 67% (6/9) of the developers	Information **not needed**

Table 4: *UI, Task or Software-Components additional information needed by the developers during the different steps of the composition process.*

different degrees; developers must have the control of the three models; they need to visualize and manipulate these models when needed. However, to achieve the composition, developers did not need to manipulate the code of the resulting application.

From a system development point of view, further work will be orientated toward making the management of the three models by the developers as fluent and appropriate as possible. To elaborate new specifications for the system (especially for determining the strict level of information necessary for composing), we will further exploit the comments developers made during the initial user testing. Iterative user testing of the next versions of the system will be performed. Initially interested in designing the OntoCompo approach and system for developers, we will consider to design them for end users too, so contributing to the End-User Development trend.

REFERENCES

Alves, A. et al. (2007), Web Services Business Process Execution Language Version 2.0, Technical report, OASIS Web Services Business Process Execution Language (WSBPEL) TC.

Bourguin, G., Lewandowski, A. and Tarby, J.-C. (2007), Defining task oriented components, *in* 'Task Models and Diagrams for User Interface Design', Springer, pp. 170–183.

Brel, C., Pinna-Déry, A.-M., Renevier, P. and Riveill, M. (2011), OntoCompo: A Tool To Enhance Application Composition, *in* '13th IFIP TC13 Conference in Human-Computer Interaction INTERACT 2011(Interact 2011)', , pp. 588–591.

Brel, C., Renevier, P., Pinna-Déry, A.-M. and Riveill, M. (2012), Annotated Component-Based Description for Application Composition, *in* 'The Seventh International Conference on Software Engineering Advances (ICSEA 2012)'.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V. and Stefani, J.-B. (2006), 'The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems', *Softw. Pract. Exper.* **36**(11-12), 1257–1284.

Corby, O., Gaignard, A., Faron-Zucker, C., Montagnat, J. et al. (2012), Kgram versatile inference and query engine for the web of linked data, *in* 'Proceedings of the International Conference on Web Intelligence', pp. 1–8.

Criado, J., Padilla, N., Iribarne, L. and Asensio, J.-A. (2010), User interface composition with cots-ui and trading approaches: Application for web-based environmental information systems, *in* 'Knowledge Management, Information Systems, E-Learning, and Sustainability Research', Springer, pp. 259–266.

et al., M. B. (2005), 'Service component architecture - building systems using a service oriented architecture', *A Joint Whitepaper by BEA, IBM, Interface21, IONA, SAP, Siebel, Sybase* .

Gabillon, Y., Petit, M., Calvary, G., Fiorino, H. et al. (2011), Automated planning for user interface composition, *in* 'Proceeding of the 2nd SEMAIS workshop of the IUI 2011 conference'.

Ginzburg, J., Rossi, G., Urbieta, M. and Distante, D. (2007), Transparent interface composition in web applications, *in* 'Proceedings of the 7th international conference on Web engineering', Springer-Verlag, pp. 152–166.

Hourdin, V., Tigli, J.-Y., Lavirotte, S., Rey, G. and Riveill, M. (2008), Slca, composite services for ubiquitous computing, *in* 'Proceedings of the International Conference on Mobile Technology, Applications, and Systems', Mobility '08, ACM, New York, NY, USA, pp. 11:1–11:8.

Joffroy, C., Caramel, B., Dery-Pinna, A.-M. and Riveill, M. (2011), When the functional composition drives the user interfaces composition: process and formalization, *in* 'Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems', EICS '11, ACM, New York, NY, USA, pp. 207–216.

Khalaf, R., Mukhi, N. and Weerawarana, S. (2003), Service-Oriented Composition in BPEL4WS, *in* 'Proceedings of the 20th Internation World Wide Web Conference (Alternate Papers Track)', WWW'03, Budapest, Hungary.

Lau, K.-K. and Rana, T. (2010), A taxonomy of software composition mechanisms, *in* 'Proc. 36th EUROMICRO Conference on Software Engineering and Advanced Applications', IEEE, pp. 102–110.

Lepreux, S. and Vanderdonckt, J. (2007), Towards a support of user interface design by composition rules, *in* 'Computer-Aided Design of User Interfaces V', Springer, pp. 231–244.

Lepreux, S., Vanderdonckt, J. and Kolski, C. (2010), User Interface Composition with UsiXML, *in* 'Proceedings of the 1st Int. Workshop on User Interface Extensible Markup Language', Berlin, Germany, pp. 141–151.

Lewandowski, A., Lepreux, S. and Bourguin, G. (2007), Tasks models merging for high-level component composition, *in* 'Human-Computer Interaction. Interaction Design and Usability', Springer, pp. 1129–1138.

Lieberman, H., Patern, F., Klann, M. and Wulf, V. (2006), End-user development: An emerging paradigm, *in* H. Lieberman, F. Patern and V. Wulf, eds, 'End User Development', Vol. 9 of *Human-Computer Interaction Series*, Springer Netherlands, pp. 1–8.

Luyten, K., Vandervelpen, C. and Coninx, K. (2002), Migratable user interface descriptions in component-based development, *in* P. Forbrig, Q. Limbourg, B. Urban and J. Vanderdonckt, eds, 'DSV-IS', Vol. 2545 of *Lecture Notes in Computer Science*, Springer, pp. 44–58.

Mehta, N. R., Medvidovic, N. and Phadke, S. (2000), Towards a taxonomy of software connectors, *in* 'Proceedings of the 22nd international conference on Software engineering', ICSE '00, ACM, New York, NY, USA, pp. 178–187.

Monk, A. (1993), *Improving Your Human-Computer Interface: A Practical Technique*, The BCS Practitioner Series, Prentice Hall.

Nestler, T., Feldmann, M., Preuner, A. and Schill, A. (2009), Service composition at the presentation layer using web service annotations, *in* 'Proceedings of the 1st Intl. Workshop on Lightweight Integration on the Web', San Sebastian, Spain, pp. 63–68.

Ope (2007), *SCA Service Component Architecture - Assembly Model Specification*. Version 1.00.

Paternò, F., Mancini, C. and Meniconi, S. (1997), Concurtasktrees: A diagrammatic notation for specifying task models, *in* 'Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction', INTERACT '97, Chapman & Hall, Ltd., London, UK, UK, pp. 362–369.

Paternò, F., Santoro, C. and Spano, L. D. (2009), 'Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments', *ACM Trans. Comput.-Hum. Interact.* **16**(4), 19:1–19:30.

Paternò, F., Santoro, C. and Spano, L. D. (2011), 'Engineering the authoring of usable service front ends', *J. Syst. Softw.* **84**(10), 1806–1822.

Pietschmann, S., Voigt, M., Rümpel, A. and Meißner, K. (2009), Cruise: Composition of rich user interface services, *in* 'Web Engineering', Springer, pp. 473–476.

Sohrabi, S. and McIlraith, S. A. (2010), Preference-based web service composition: A middle ground between execution and search, *in* 'Proceedings of the 9th International Semantic Web Conference (ISWC-10)', Shanghai, China, pp. 713–729.

Stuerzlinger, W., Chapuis, O., Phillips, D. and Roussel, N. (2006), User Interface Façades: Towards Fully Adaptable User Interfaces, *in* 'UIST '06: ACM Symposium on User Interface Software and Technology', ACM - SIGCHI & SIGGRAPH, ACM, Montreux, Suisse, pp. 309–318.

Tan, D. S., Meyers, B. and Czerwinski, M. (2004), Wincuts: manipulating arbitrary window regions for more effective use of screen space, *in* 'CHI'04 extended abstracts on Human factors in computing systems', ACM, pp. 1525–1528.

Vermeulen, J., Vandriessche, Y., Clerckx, T., Luyten, K. and Coninx, K. (2007), Service-interaction descriptions: Augmenting services with user interface models, *in* J. Gulliksen, M. B. Harning, P. A. Palanque, G. C. van der Veer and J. Wesson, eds, 'EHCI/DS-VIS', Vol. 4940 of *Lecture Notes in Computer Science*, Springer, pp. 447–464.

Zhao, Q., Huang, G., Huang, J., Liu, X. and Mei, H. (2008), A web-based mashup environment for on-the-fly service composition, *in* 'Service-Oriented System Engineering, 2008. SOSE'08. IEEE International Symposium on', IEEE, pp. 32–37.